International Journal of Engineering Sciences & Research Technology

(A Peer Reviewed Online Journal)
Impact Factor: 5.164

IJESRT

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

IMPLEMENTATION OF AUTOENCODER ON MNIST HANDWRITTEN DIGITS

Ranganadh Narayanam
Department of Electronics and Communications Engineering, Faculty of Science and Technology, The ICFAI Foundation for Higher Education, Hyderabad, (Deemed to be University under section 3 of UGC Act, 1956), India

DOI: https://doi.org/10.29121/ijesrt.v10.i2.2021.5

ABSTRACT

Autoencoders (AE) are a family of neural networks for which the input is the same as the output. They work by compressing the input into a latent-space representation and then reconstructing the output from this representation. The aim of an Autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal “noise”. In this paper De-noising Autoencoder is implemented by proposing a novel approach on MNIST handwritten digits. This model is validated through training and validation losses, and observing the reconstructed test images when comparing to the original images. The proposed model is found to be working very well.

1. INTRODUCTION

Autoencoder [1] is a type of neural network where the output layer has the same dimensionality as the input layer. In simpler words, the number of output units in the output layer is equal to the number of input units in the input layer. An autoencoder replicates the data from the input to the output in an unsupervised manner and is therefore sometimes referred to as a replicator neural network. "Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) learned automatically from examples rather than engineered by a human. The autoencoders reconstruct each dimension of the input by passing it through the network. It may seem trivial to use a neural network for the purpose of replicating the input, but during the replication process, the size of the input is reduced into its smaller representation. The middle layers of the neural network have a fewer number of units [3] as compared to that of input or output layers. Therefore, the middle layers hold the reduced representation of the input. The output is reconstructed from this reduced representation of the input. Autoencoders are similar in spirit to dimensionality reduction techniques like principal component analysis. They create a space where the essential parts of the data are preserved, while non-essential (or noisy) parts are removed.

Autoencoders architecture

As depicted in the Figure 1, there are three components in an Autoencoder : Encoder, Code, Decoder.

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

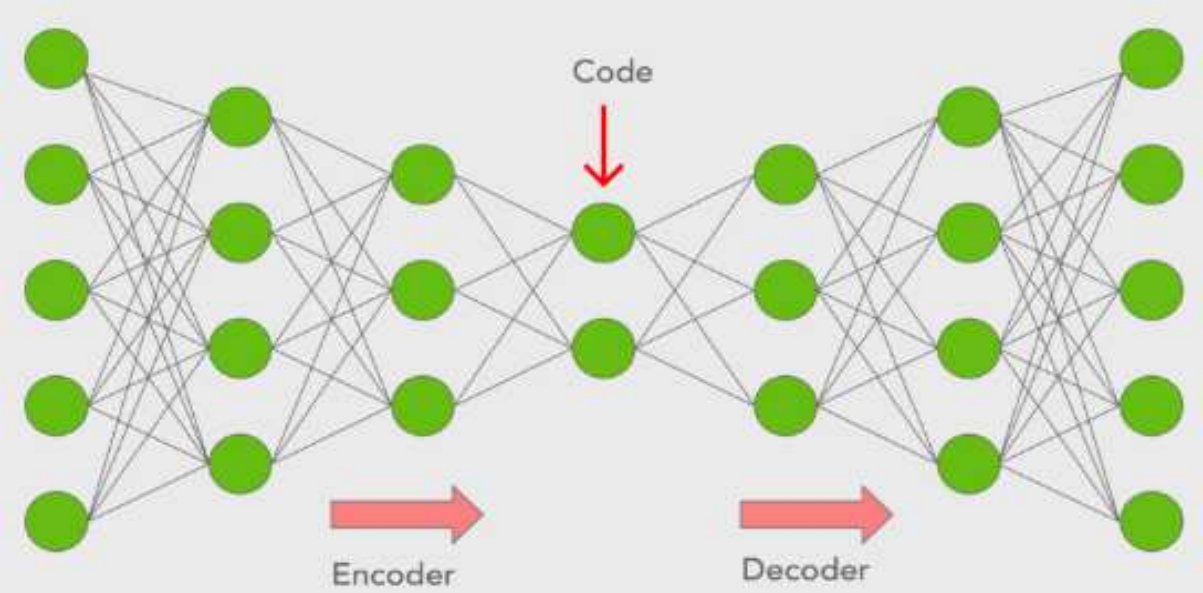

Figure 1: Architecture of Autoencoder

Encoder
This component is a feedforward, fully connected neural network. Its functionality is to compress the input into a latent space representation, and encodes the input image into a compressed representation in a reduced dimension. Distorted version of the original image is obtained after compression.

Code
This part of the network contains the reduced representation of the input that is fed into the decoder.

Decoder
It is also a feedforward network and has a similar structure to the encoder. From the code this network reconstructs the input back to the original dimensions. First, the input goes through the encoder where it is compressed and stored in the layer called Code, then the decoder decompresses the original input from the code.

The main objective of the autoencoder is to get an output identical to the input. The decoder architecture is the mirror image of the encoder, typically, but not a requirement. The dimensionality of the input and output must be the same is the only requirement.

Some important varieties of Autoencoders are Convolutional Autoencoder, Denoising Autoencoder [1], Variational Autoencoder. Here in this research we have considered to implement De-noising autoencoder.

Denoising autoencoder
In this research Denoising Autoencoder is implemented using Tensorflow(Python) using MNIST handwritten digits. It is to learn a representation (latent space) that is robust to noise is the idea behind a Denosing Autoencoder [3]. We add noise to an image and then feed this noisy image as an input to our network. Here, The encoder part transforms the image into a different space that preserves the handwritten digits but removes the noise. The original image is 28 x 28 x 1 image, and the transformed image is 7 x 7 x 32. We can think of the 7 x 7 x 32 image as a 7 x 7 image with 32 color channels. The decoder part of the network then reconstructs the original image from this 7 x 7 x 32 image by removing the noise. During training, we define a loss (cost function) to minimize the difference between the reconstructed image and the original noise-free image, means we learn a 7 x 7 x 32 space that is noise free.

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

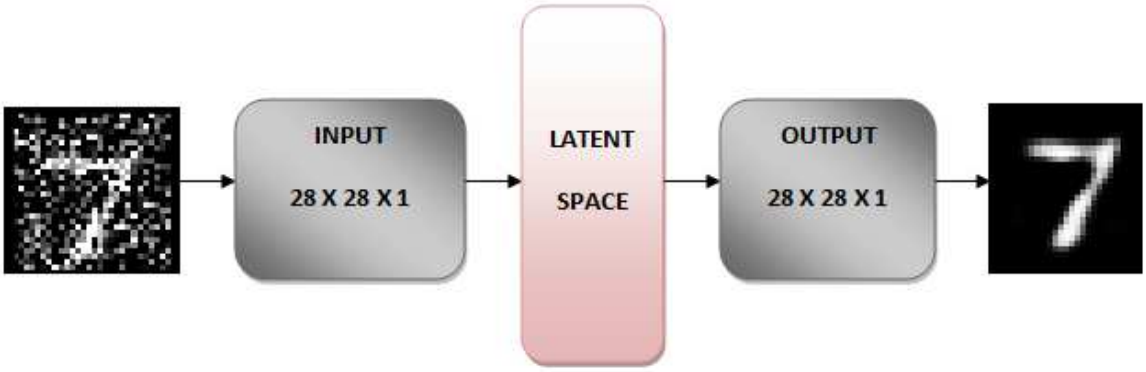

Figure 2: Denoising Autoencoder

2. IMPLEMENTATION OF DENOISING AUTOENCODER

The network

The images are matrices of size 28 x 28. We reshape the image to be of size 28 x 28 x 1, convert the resized image matrix to an array, rescale it between 0 and 1, and feed this as an input to the network. The encoder transforms the 28 x 28 x 1 image to a 7 x 7 x 32 image. You can think of this 7 x 7 x 32 image as a point in a 1568 (because 7 x 7 x 32 = 1568) dimensional space. This 1568 dimensional space is called the bottleneck or the latent space. The architecture is graphically shown in Figure 3. The decoder does the exact opposite of an encoder; it transforms this 1568 dimensional vector back to a 28 x 28 x 1 image. We call this output image a "reconstruction" of the original image. The structure of the decoder is shown in Figure 4.

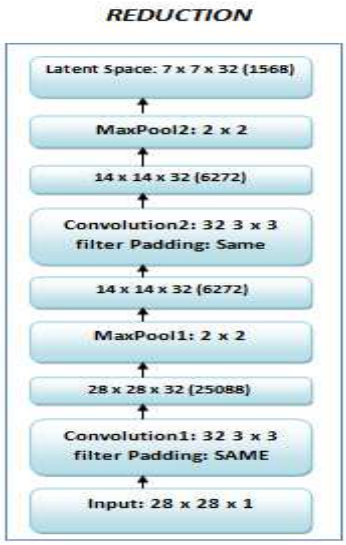

Figure 3: Architecture of encoder model

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

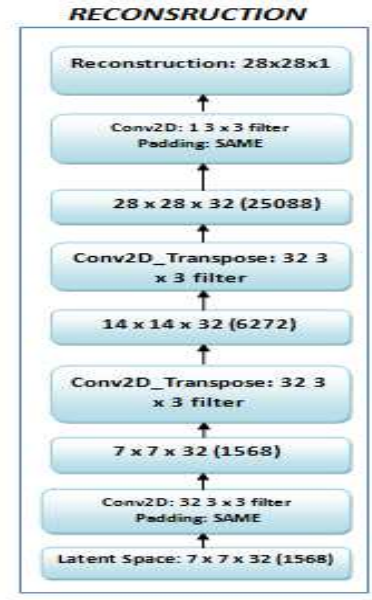

Figure 4: Architecture of decoder model

The following sections describe the implementation of Denoising Autoencoder using Tensorflow.

Encoder

There are 2 Convolutional layers and 2 max pooling layers. Both layer-1 and layer-2 of Convolution have 32-3 x 3 filters. There are two max-pooling layers each of size 2 x 2.

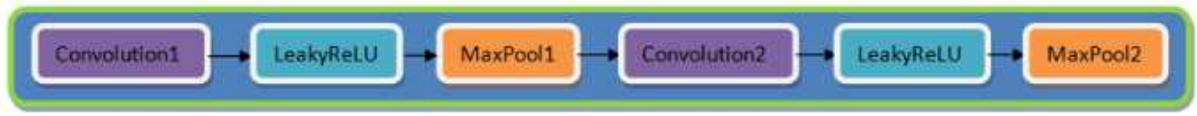

Figure 5: Encoder block diagram

Decoder

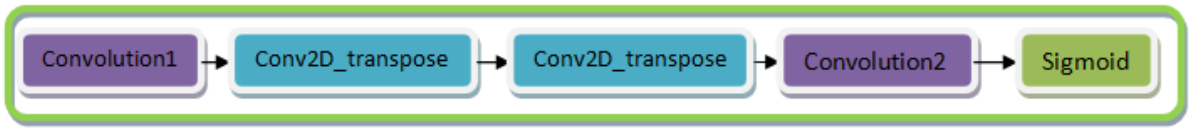

Figure 6: Decoder block diagram

The decoder has two Conv2d_transpose layers, two Convolution layers, and one Sigmoid activation function. Conv2d_transpose is for upsampling which is opposite to the role of a convolution layer. The Conv2d_transpose layer upsamples the compressed image by two times each time we use it. Finally, we calculate the loss of the output using cross-entropy loss function and use Adam optimizer to optimize our loss function.

USING Leaky ReLU INSTEAD of ReLU as ACTIVATION FUNCTION

We want gradients to flow while we backpropagate through the network. We stack many layers in a system in which there are some neurons whose value drop to zero or become negative. Using a ReLU as an activation function clips the negative values to zero and in the backward pass, the gradients do not flow through those neurons where the values become zero. Because of this the weights do not get updated, and the network stops learning for those values. So using ReLU is not always a good idea. Therefore, we use a leaky ReLU which instead of clipping the negative values to zero, cuts them to a specific amount based on a hyper parameter alpha. This ensures that the network learns something even when the pixel value is below zero.

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

Load the data

Once the architecture has been defined, we load the training and validation data. As shown below, Tensorflow allows to easily load the MNIST data. The training and testing data loaded is stored in variables train_imgs and test_imgs respectively. Since its an unsupervised task no need to care about the labels.

 # load mnist dataset
(train_imgs, train_labels), (test_imgs, test_labels) = tf.keras.datasets.mnist.load_data()
 # fit image pixel values from 0 to 1
train_imgs, test_imgs = train_imgs / 255.0, test_imgs / 255.0

Data analysis

Before training a neural network, it is always a good idea to do a sanity check on the data. The data consists of handwritten numbers ranging from 0 to 9, along with their ground truth labels. It has 55,000 train samples and 10,000 test samples. Each sample is a 28×28 grayscale image.

The data details

 # check data array shapes:
print("Size of train images: {}, Number of train images: {}".format(train_imgs.shape[-2:], train_imgs.shape[0]))
print("Size of test images: {}, Number of test images: {}".format(test_imgs.shape[-2:], test_imgs.shape[0]))

The output is

Size of train images: (28, 28), Number of train images: 60000
Size of test images: (28, 28), Number of test images: 10000

The visualization of train and test image examples

```
# plot image example from training images

plt.imshow(train_imgs[1], cmap='Greys')
plt.show()
# plot image example from test images
plt.imshow(test_imgs[0], cmap='Greys')
plt.show()
plt.close()
```

Output:

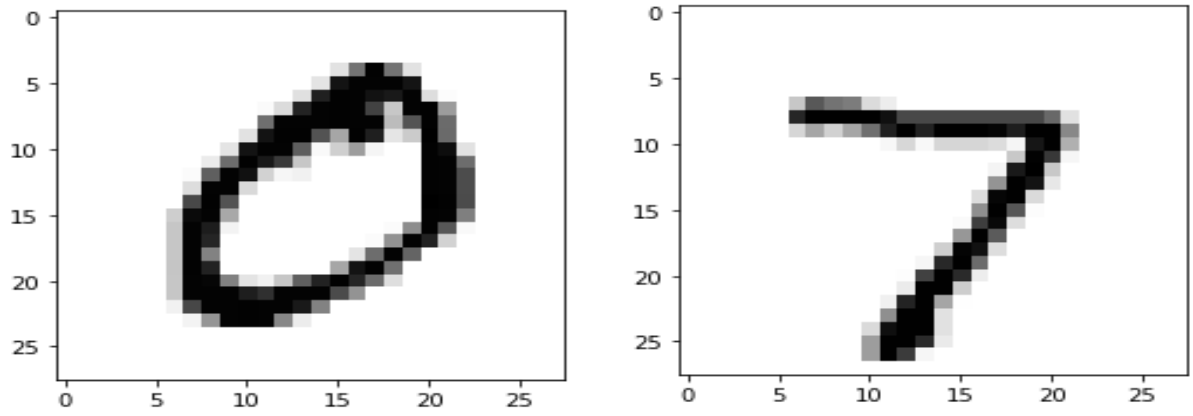

Figure 7: Train and test MNIST images

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

Pre-processing data

The images are greyscale and the pixel values range from 0 to 255. We apply following pre-processing to the data before feeding it to the network.

1. Add a new dimension to the train and test images, which will be fed into the network.

```
# prepare training reference images: add new dimension
train_imgs_data = train_imgs[..., tf.newaxis]
# prepare test reference images: add new dimension
test_imgs_data = test_imgs[..., tf.newaxis]
```

2. Add noise to both train and test images which we then feed into the network. Noise factor is a hyperparamter and can be tuned accordingly.

```
# add noise to the images for train and test cases
def distort_image(input_imgs, noise_factor=0.5):
    noisy_imgs = input_imgs + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=input_imgs.shape)
    noisy_imgs = np.clip(noisy_imgs, 0., 1.)
    return noisy_imgs
# prepare distorted input data for training
train_noisy_imgs = distort_image(train_imgs_data)
# prepare distorted input data for evaluation
test_noisy_imgs = distort_image(test_imgs_data)
```

3. Noisy images illustration

```
# plot distorted image example from training images
image_id_to_plot = 0
plt.imshow(tf.squeeze(train_noisy_imgs[image_id_to_plot]), cmap='Greys')
plt.title("The number is: {}".format(train_labels[image_id_to_plot]))
plt.show()
# plot distorted image example from test images
plt.imshow(tf.squeeze(test_noisy_imgs[image_id_to_plot]), cmap='Greys')
plt.title("The number is: {}".format(test_labels[image_id_to_plot]))
plt.show()
plt.close()
```

OUTPUT

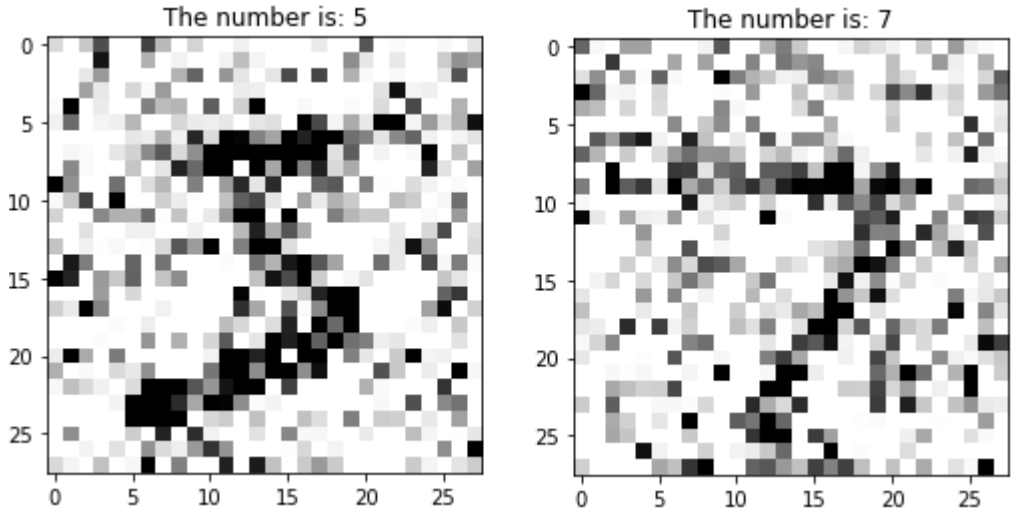

Figure 8: Noisy train and test MNIST images

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

Train and evaluate the model

The network is ready to get trained. The number of epochs specified as 25 and batch size of 64, means the whole dataset will be fed to the network 25 times. We will be using the test data for validation. After 25 epochs we can see our training loss and validation loss is quite low which means our network is performing well.

Training vs. validation loss plot

loss plot drawn between training and validation data using the introduced utility function plot_losses(results).

```
# funstion for train and val losses visualizations
def plot_losses(results):
    plt.plot(results.history['loss'], 'bo', label='Training loss')
    plt.plot(results.history['val_loss'], 'r', label='Validation loss')
    plt.title('Training and validation loss',fontsize=14)
    plt.xlabel('Epochs ',fontsize=14)
    plt.ylabel('Loss',fontsize=14)
    plt.legend()
    plt.show()
    plt.close()

# visualize train and val losses

plot_losses(results)
```

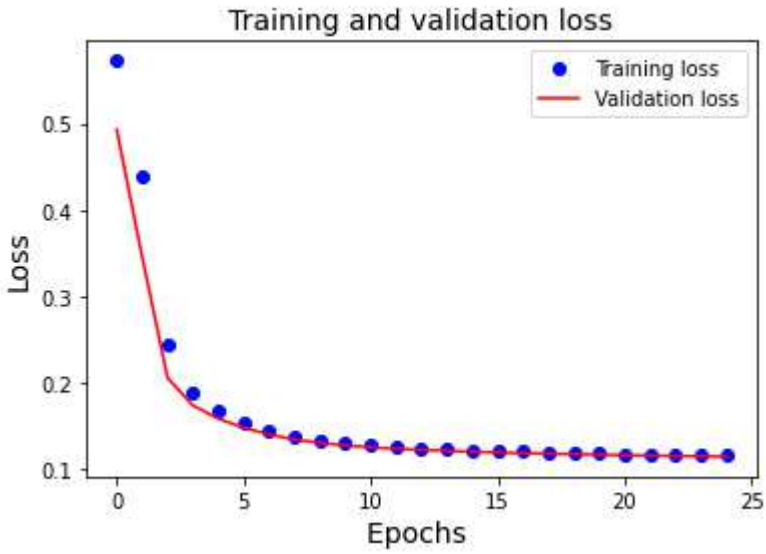

Figure 9: Training and validation losses

It is evident from the loss plot that in the first 10 epochs, validation loss and training loss are both steadily decreasing. This training loss and the validation loss are also very close to each other. This means that our model has generalized well to unseen test data. We can further validate our results by observing the original, noisy and reconstruction of test images.

ISSN: 2277-9655
Impact Factor: 5.164
CODEN: IJESS7

3. RESULTS

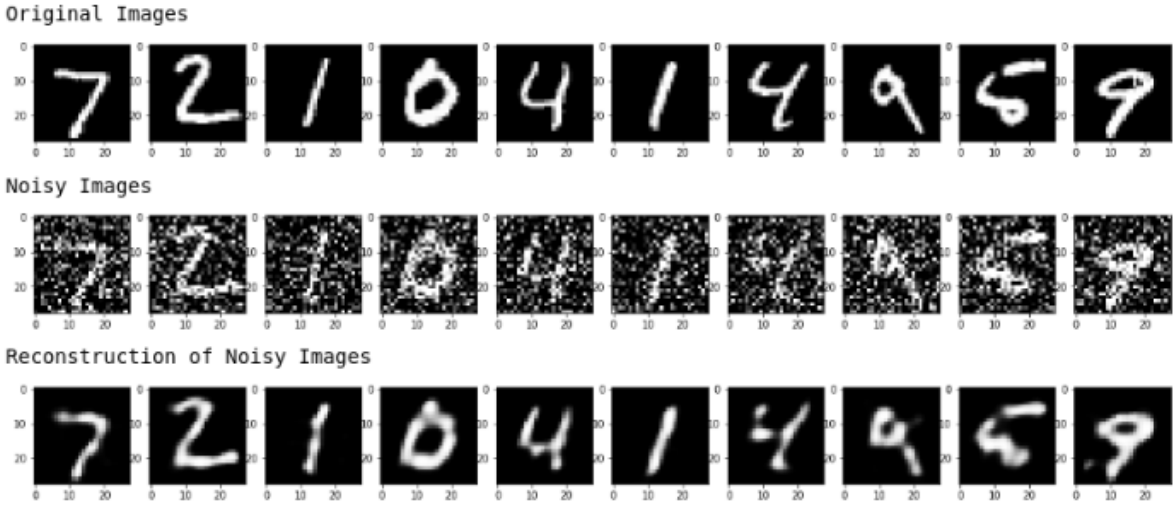

Figure 10: Representation of MNIST images on different stages

4. CONCLUSION

In this research we developed new approach for implementing the Denoising Autoencoder. We implemented it on MNIST handwritten digits. Using this proposed model we can be able to reconstruct the images with good precision. Some parts of the programs also addressed. It is evident from the "Figure 9: Training and validation losses" plot; and then the reconstructed images in "Figure 10: Representation of MNIST images on different stages" that our proposed model is working well and is applicable novel approach for implementing Denoising Autoencoder

REFERENCES

[1] "Auto-Encoder Variants for Solving Handwritten Digits Classification Problem", Muhammad Aamir, Nazri Mohd Nawi2 Hairulnizam Bin Mahdin1 Rashid Naseem3 and Muhammad Zulqarnain, International Journal of Fuzzy Logic and Intelligent Systems 2020;20(1):8-16 Published online March 25, 2020, 2020 Korean Institute of Intelligent Systems.

[2] "Generalized Denoising Auto-Encoders as Generative Models", Y.Bengio, Li yao, G Alian, Advances in Neural Information Processing Systems, 2013

[3] "A Deep Auto-Encoder based Approach for Intrusion Detection System", Fahimeh Farahnakian, Jukka Heikkonen, International Conference on Advanced Communications Technology(ICACT), 2018.

[4] "A Novel Framework Using Deep Auto-Encoders Based Linear Model for Data Classification", Ahmad M. Karim,Hilal Kaya,Mehmet Serdar Güzel,Mehmet R. Tolun,Fatih V. Çelebi,and Alok Mishra, MDPI Sensors (Basel). 2020 Nov; 20(21): 6378